Parameterized Packages and Java

Débora Aranha and Paulo Borba*
Departamento de Informática
Universidade Federal de Pernambuco

Abstract

This paper proposes an extension of Java with support for parameterized *packages*. This significantly enhances Java's capabilities for software reuse and maintenance *in the large*. We discuss several approaches for translating the proposed extension into Java, showing that it can be implemented in a relatively efficient way. We also justify our design decisions, comparing them with proposals for extending Java with support for parameterized *classes*.

Resumo

Neste trabalho nós apresentamos uma extensão de Java com suporte para *pacotes* parametrizados. Assim aumenta-se consideravelmente o poder de Java para reuso e manutenção de software em larga escala. Além disso, nós discutimos várias abordagens para implementar, de forma relativamente eficiente, a extensão apresentada traduzindo-a para Java. Nós também justificamos as nossas decisões de projeto, fazendo uma comparação com outras abordagens para extensão de Java com *classes* parametrizadas.

1 Introduction

Object-oriented programming is widely known for its support for software reuse and maintenance. This support is based on powerful concepts for structuring programs: objects and classes, subtype polymorphism, inheritance, and dynamic binding [15].

Packages can enhance the support for reuse and maintenance offered by object-oriented languages [11, 10, 3, 7]. In fact, by encapsulating the definitions of related classes, packages become essential for structuring complex object-oriented systems and, consequently, supporting large scale reuse and maintenance. The need for both classes and packages in the same language is now widely recognized, and both concepts are available in several modern object-oriented languages, including Java [12].

*Supported in part by CNPq, Grant 301021/95-3. Email: {`dcsa,phmb`}`@di.ufpe.br`. WWW: `http://www.di.ufpe.br/~{dcsa,phmb}`. Address: Caixa Postal 7851, CEP 50732-970, Recife, PE, Brazil.

Parametric polymorphism is another powerful concept for enhancing reuse and maintenance. Whereas subtype polymorphism makes code generic across a family of related types[1], parametric polymorphism makes code generic across unrelated types [6]. For example, since a list of bank clients behaves in much the same way as a list of bank accounts, it is not reasonable to write virtually identical code for both lists. With parametric polymorphism, programmers can simply write a parameterized (generic) code for lists. This code can then be instantiated and used for clients, accounts, and many other types, even though there is no subtyping relationship between them.

The kind of genericity mentioned above could also be achieved by writing a virtually untyped code for (heterogeneous) lists; in Java, for instance, we would specify that lists should store objects of `Object`, a superclass of every other class. Types `Vector` and `HashTable` defined in a standard Java library (`java.util`) are implemented in this way. So the same vector could store strings and integers! Consequently, programmers must use type *casts* to discriminate types. This seriously compromises safety and forces programmers to write awkward code. Hence several object-oriented languages, notably Eiffel [15] and C++ [18], support parametric polymorphism. Unfortunately, the current specification of Java [12] offers no such support, but some extensions in that direction have already been proposed [16, 17].

In Eiffel, C++, and several other languages, including the extensions of Java mentioned above, *classes* are parameterized by *classes*[2]. For example, a class defining lists is parameterized by the class of the elements to be included in the lists. This is a natural choice for supporting parametric polymorphism in languages having classes only. However, is that the best choice for languages supporting both classes and packages? Or is it just one more unfortunate consequence of neglecting until recently the need for both mechanisms?

For favouring effective software reuse and maintenance, we suggest that object-oriented languages such as Java, having both classes and packages, should support parameterized polymorphism at the level of packages only. That is, *packages* should be parameterized by *packages*. FOOPS [9, 2] first suggested this approach, using large-grain structuring concepts originated from Clear [4] and later implemented in OBJ [8] and, partially, in Standard ML [14], Ada and other languages.

Based on the design of FOOPS, we propose a conservative extension of Java, called PJava, with support for parameterized *packages*. We claim that Java's capabilities for reuse and maintenance *in the large* can be significantly enhanced in this way. Furthermore we argue that extensions of Java with support for parameterized *classes* [16, 17] enhance only programming *in the small*, in a way that can be naturally achieved with PJava features for programming *in the large*.

This paper is organized as follows. First we informally present the features of PJava that extend Java. The design of PJava is quite conservative: we tried to minimize modifications, simplify extensions, and reduce impact on others aspects of Java. In this way we hope that Java programmers can easily adapt to PJava. Second we discuss several approaches for translating PJava into Java. This details PJava's operational behaviour and shows that PJava can be implemented in a relatively

[1] Related with respect to subtyping.

[2] To be more precise, in the extensions of Java, *classes* and *interfaces* [12] are parameterized by Java *types* (primitive data types, classes, and interfaces).

efficient way. As a consequence of this translation, PJava programs can normally interoperate with plain Java programs and class libraries; also, as Java translates to the Java Virtual Machine (JVM) [13], PJava code is available for execution in a myriad of platforms. Finally we justify PJava design decisions, comparing them with alternative approaches.

2 Introducing PJava

In this section we introduce PJava through a simple, yet complete, example. PJava is a conservative extension of Java, with just a few extensions to the language. Here we concentrate on those extensions; other aspects of Java are explained elsewhere [12].

2.1 Package Types

PJava extends Java with parameterized packages, which are also called generic packages. A generic package looks much the same as a (non generic) package, except that it is parameterized by other packages, and so can refer to those parameters in its body. In order to be used, generic packages must be instantiated with actual arguments (packages). Each instantiation yields a new package. So a generic package actually defines a set of related packages, one for each possible instantiation. Packages in such a set offer similar behaviour, but differ on the type of information that can be manipulated.

However, a generic package usually must imposes some restrictions on its parameters, in much the same way that a typed method or routine impose type constraints on its parameters. For example, let us consider a generic package defining a table mapping keys to values. The types of keys and values should be defined by arguments given to this package. But, unless we can compare keys for equality, we cannot implement a method for finding the value associated to a given key. So valid arguments should provide not only an implementation for the type of keys, but also to an associated method for checking equality.

In PJava, package parameters have associated *package types* for specifying which arguments are valid. A package type is similar to a Java package, except that it only groups the definitions of related interfaces: types defined by the signature of its operations [12]. In package types, interfaces specify the constraints that must be satisfied by valid arguments. For example, parameters having the following package type

```
package type KEY;
interface Key {
  boolean equals(Key k);
  boolean lt(Key k);
}
```

can only be instantiated with packages having a class that offers at least the same functionality as `equals` and `lt`.

When there are no constraints on parameters, we use a package type such as the following:

```
package type VALUE;
interface Value {}
```

In more complex examples, a package type usually contains several dependent interfaces. This indicates that a package parameter actually needs to impose constraints on a group of (mutually) dependent concepts. For the sake of modularization, it is important to package together those concepts. Mutual dependencies are often useful as well:

```
package type GRAPH;
interface Node { Edge[] edges(); ...}
interface Edge { Node source(); ...}
```

Package types can also state subtype relationships between its components. For instance, that would be useful when defining a class `Bank` parameterized by the types of accounts that it can handle: current and saving accounts. In fact, we normally need to say that `SavingAcct` is a subtype of `CurrentAcct`:

```
package type ACCTS;
interface CurrentAcct { void credit(double v); ...}
interface SavingAcct extends CurrentAcct { void savings(); ...}
```

2.2 Generic Packages

A generic package is very similar to a Java package, except that it has one or more typed package parameters. For example, let us progressively introduce a generic package `TABLE` defining a class of tables mapping keys to values:

```
generic package TABLE[KEY K, VALUE V];
public class Table {
  private Key[] keys; private Value[] values;
  private int size, maxsize;
  public Table(int max){
    maxsize = max; keys = new Key[maxsize];
    size = 0; values = new Value[maxsize];
  }
```

First notice that the package body can refer not only to the *package variables* `K` and `V`, but also to the entities that must be associated to them at instantiation time; that is, the entities defined in the package types `KEY` and `VALUE`. In case of naming conflicts, the types `Key` and `Value` could be respectively accessed using `K.Key` and `V.Value`.

The class `Table` has a method for inserting an indexed value in a table, and another for recovering a value associated to a given key. In order to implement those methods, we can refer to the types `Key` and `Value`, but the only operations allowed on their objects are precisely those listed in their associated package types (`KEY` and `VALUE`):

```
  public void insert(Key k, Value v) throws FullTableException {
    int i = 0;
    while (i < size && keys[i].lt(k)) i++;
    if ... { ... keys[i] = k; values[i] = v; ...}
    else { throw new FullTableException(); }
  }
  public Value recover(Key k) throws NotFoundException {
    ... throw new NotFoundException(k); ... }
} \\ end of class Table
```

Note that two exception types have yet to be defined in the generic package:

```
public class FullTableException extends Exception {...}
public class NotFoundException extends Exception {
  private Key key;
  NotFoundException (Key k) {key = k;}
  public Key key(){return key;}
  ...
}
```

One of them has a variable of type `Key` so that its exceptions can register which key caused the failure.

As can be observed from the example above, generic packages are much the same as non generic packages. Indeed, any entity declared in a non generic package can be declared in a generic package. This means that generic packages can import other packages, including instantiations of generic packages, as we will see later.

2.3 Instantiating Generic Packages

Generic packages are instantiated with arguments. Each instantiation yields a new package having the classes and interfaces declared in the generic package, but adapted to store and manipulate objects of the types defined in the packages given as arguments. This new package may be used as an ordinary Java package.

Let us illustrate how the generic package `TABLE` could be instantiated in PJava. Suppose we wish to obtain a table mapping integer codes to strings, where codes are defined as follows:

```
package CODE;
public class Code {
  private int code;
  public Code(int n) {this.code = n;}
  public int code() {return this.code;}
  public boolean eq(Code c) {return this.code == c.code();}
  public boolean neq(Code c) {return ! this.eq(c);}
}
```

and the class `String` of strings is defined in the package `java.lang` [12]. So we should instantiate `TABLE` with the packages `CODE` and `java.lang`. But, as the package `java.lang` has several classes, we need to say precisely which one shall be associated to Value. Similarly, as Code has several methods, we need to specify which ones shall provide the functionality of equals and lt. We could also say that Code shall be associated to Key, but a compiler can easily infer that since CODE contains only one class.

It is then clear that a package may satisfy a package type in many ways. PJava supports *views* for indicating how a package should satisfy a package type in a particular instantiation. Essentially, a view is a mapping from entities names and expressions into entities names and expressions, indicating how packages should match package types. For example, in order to obtain a table mapping codes to strings, we should instantiate TABLE using the following views:

```
TABLE[CODE {  Key -> Code;
              equals -> eq;
              Key k1, k2; k1.lt(k2) -> k1.code() < k2.code()
            },
      java.lang {Value -> String}]
```

Note that lt is not mapped into a method of Code, but into an expression formed by methods of Code. This is quite useful since, in our case, Code does not have a suitable method for matching lt, but it has the potential to implement such a method. Also, the mapping "equals -> eq" is just an abbreviation for the mapping

```
boolean equals(Key k) -> boolean eq(Code c)
```

which should be used in case of naming conflicts. The mappings that can be inferred by a compiler may be omitted, and in some cases we may assume default views [9, 10, 8]; we omit the details here.

We consider that different instantiations of a generic package yield different packages unless equivalent views are used in the instantiations. Two views are equivalent if, independent of variable names and order of appearance, they contain the same binding components. This straightforward approach for equivalence neither reduces views' expressive power nor compromises static type checking[3].

Instantiations of generic packages provide adapted code for implementing other packages, either by being used for further instantiations or by being imported:

```
package TEST;
import TABLE[CODE { Key -> Code; ...},
             java.lang {Value -> String}].*;
```

As in Java, the symbol '*' after the package instantiation indicates that all public entities of that package should be imported.

Inside TEST, we can then write the following test code:

```
Table info = new Table(50); Code c = new Code(10);
info.insert(c,"PJava"); info.insert(new Code(1),"Java");
try { String s = info.recover(c);
      System.out.println(s + " is a nice language...");
} catch (NotFoundException e) {
   System.out.print("There is no string for " + e.key().code());
}
```

[3] Contrast with an alternative approach described in [1].

Instantiations can also be done in the body of generic packages. For example, a declaration such as

```
import LIST[java.lang {Elt -> Integer}].*;
```

could well be in the generic package `TABLE`, if an auxiliary list of integers were necessary. Instantiations in the body of a generic package can actually be performed with the formal parameters of that package. For example, the package `TABLE` could have the declaration

```
import LIST[K {Elt -> Key}].*;
```

so that a list, instead of an array, is used to store keys. This instantiation is valid because the type of `K` offers more functionality than the type of the parameter of `LIST`, which only requires a type `Elt`.

2.3.1 Compatibility Rules

An instantiation is valid only if the arguments provided to parameter packages offer at least the same functionality as their associated types. In order to guarantee that, valid views must satisfy some compatibility rules. Those rules basically assure that the entities declared in the types of parameter packages are mapped to "compatible" entities declared in argument packages.

A view `v` can arbitrarily map type names. However, it is only valid if it preserves the subtype notion of package types; that is, if `T` is a subtype of `T'`, then `v(T)`—the image of `T` under `v`—must be a subtype of `v(T')`. Also methods declared in a type `T` must be mapped into methods declared in a supertype of `v(T)`[4]. Furthermore, if a method yields results (exceptions) of type `R`, it must be mapped into a method that yields results (exceptions) of subtypes of `v(R)`. Finally, parameterized methods have to be mapped into methods with the same number of parameters, but a parameter of type `T` must correspond to a parameter of a supertype of `v(T)`.

Similar rules applies for mapping of expressions. In summary, if the type of the expression on the left is `T` then the expression on the right must be of a subtype of `v(T)`. For example, the mapping

```
Key k1, k2; k1.lt(k2) -> k1.code() < k2.code();
```

is valid because the type of the expression on the left is `boolean`, which is mapped to itself and is also the type of the expression on the right. Also, `k1` and `k2` are of type `Key`, which is mapped into `Code`, so that expression `k1.code() < k2.code()` is well typed.

2.4 Renaming

In order to adapt packages and package types to new contexts, it is useful, and sometimes necessary, to rename some declared entities. In PJava, this is performed by the renaming operator '`#`'. For example, if several instances of `TABLE` are generated in the same context we have to distinguish the several table classes created. This is done by applying the renaming operator to each instantiated package:

[4] We consider that the subtype and supertype relations are reflexive.

```
TABLE[CODE {Key -> Code; ...},
      java.lang {Value -> Integer}]
# (Table -> IntegerTable; recover -> find;
   NotFoundException -> NoMappedInt; Key key() -> Code code())
```

Note that both classes and methods can be renamed.

The renaming operator actually yields a new package, with the same functionality as its target, but with different entities names. By contrast, views establish only a temporary association between a package and a package type.

Renaming is also useful for improving reuse of package types. For instance, instead of declaring the package type KEY, it would be better to declare a package type denoting the more abstract concept of "comparable element":

```
package type CMPELT;
interface CmpElt {
  boolean equals(CmpElt e);
  boolean lt(CmpElt e);
}
```

That could be adapted to many different contexts, including table keys:

```
generic package TABLE[CMPELT # (CmpElt -> Key)  K, ...];
```

3 Translating PJava into Java

PJava is implemented by translation into Java. We consider two rather distinct approaches for translation. The heterogeneous translation generates a specialized copy of a generic package each time it is instantiated. This approach is straightforward, generates efficient code, but promotes an undesirable code expansion. The homogeneous translation generates a single copy a generic package, and this copy is shared by all instances. This approach avoids the code expansion, but the generated code is less efficient and more difficult to understand.

3.1 Heterogeneous Translation

In the heterogeneous translation package types and generic packages are not translated into Java code. They are useful for the compilation process, but have no counterparts in the generated code. Package types are used for checking whether instantiations are valid, whereas generic packages are used as templates for creating specialized Java package for each instantiation.

We translate the instantiation of a generic package into a Java package derived from the generic package by the textual substitution given by the views specified in the instantiation. For example, the translation of

```
TABLE[CODE {  Key -> Code; ...
              k1.lt(k2) -> k1.code() < k2.code()
            },
      java.lang {Value -> String}]
```

yields a new package, say TABLE_INST1, containing the declarations of TABLE, except that all references to Key and Value are respectively replaced by Code and String, and the methods and expressions are also substituted as specified in the view, as can be seen in part of the generated code for the method insert of Table:

```
public void insert(Code k, String v) throws FullTableException {
 ... while (i < size && keys[i].code() < k.code()) i++;...
}
```

The new package must also import CODE and java.lang.

Code depending on the instantiation of a generic package is normally translated to Java by replacing the instantiation text with the name of the newly created package. So, in TEST, the import statement

```
import TABLE[CODE { Key -> Code; ...},
             java.lang {Value -> String}].*;
```

is translated into import TABLE_INST1.*;.

3.2 Homogeneous Translation

Contrasting with the heterogeneous translation, in the homogeneous translation package types and generic packages are translated into Java code. In fact, generic packages are translated into Java packages that are shared by all instances of the generic package. The generated code is actually reused, instead of being duplicated as in the heterogeneous translation.

A generic package is translated into a package derived from the generic one by replacing references to types declared in parameter types by references to class Object. As Object is a superclass of any other class, the generated code can be used to manipulate objects of types used in any valid instantiation[5]. For example, consider part of the translation of TABLE:

```
package TTABLE;
public class Table {...
  public void insert(Object k, Object v) throws FullTableException
...}
```

However, note that the body of insert uses the lt method on keys, and the actual code for this method differs from one instantiation to the other. So we cannot simply translate the expression k'.lt(k) as it is.

Following an approach sketched in [16], we make sure that tables store a special object encapsulating specific code for lt. This object is provided through the table creation routine, when a compiler must know which code should be associated to lt. The method insert can then ask this object for making the necessary computations at runtime. The abstract types of these special objects are obtained by a direct translation of the types declared in associated package types. For example, the package type KEY is translated into

[5]This also works for primitive types, by wrapping them when passing as arguments, and unwrapping them when getting results.

```
package TKEY;
public interface KeyMethods {
  boolean equals(Object x, Object y);
  boolean lt(Object x, Object y);
}
```

This package is imported into `TTABLE` so that the translated class `Table` can contain the following definitions:

```
KeyMethods km;
public Table(int max, KeyMethods kmethods) {...
  km = kmethods;
}
```

and an expression such as `k'.lt(k)` can be translated to `km.lt(k',k)`, where the variable `km` stores a reference to a special object.

We translate the instantiation of a generic package into concrete types for special objects. For example, the instantiation of `TABLE` for mapping codes to strings is translated into

```
class KeyMethodsCode1 implements KeyMethods {
  public boolean lt(Object x, Object y) {
    return ((Code) x).code() < ((Code) y).code();
  }
  public boolean equals(Object x, Object y) {
    return ((Code) x).eq((Code) y);
  }
}
```

PJava code depending on that instantiation is normally translated to Java, except that import statements referring to the instantiation are translated into '`import TTABLE.*;`' and '`import TKEY.*;`', and class constructors should be provided with an extra argument, as illustrated below:

```
Table t = new Table(10, new KeyMethodsCode1());
```

The translations of arrays and object creation in generic packages are more problematic than the translations shown above. They can only be implemented in a satisfactory way using Java's advanced reflection facilities such as the class `Class`, the routine `newInstance` and the class `Array` [12]. For space reasons, we omit the details here.

4 Justifying PJava Design Decisions

In this section we justify PJava design decisions from several points of view. We concentrate on the following dilemma: parameterized *packages* or parameterized *classes*? We have also considered supporting parameterized packages and parameterized classes, but that would just introduce redundancy and unnecessary complexity to the language, as we will discuss in the following section.

4.1 Programming *in the large*

Packages and classes are complimentary concepts [10, 3, 11]. Classes group data and related operations together, defining ADTs which can be quite useful for structuring small programs. In fact, classes only support reuse and maintenance *in the small*, usually restricted to a corporation due to namespaces problems. Contrasting, packages support reuse and maintenance *in the large*, inside and across corporations, and are essential for structuring large programs consisting of many classes, since several related and dependent classes can be encapsulated in a single package.

It follows from the discussion above that, for programming in the large purposes, parameterized packages should be much more powerful than parameterized classes[6]. Indeed, it is so. Whereas a parameterized class puts classes together to compose a new class, a parameterized package puts packages together to compose a new package. But notice that each single package usually contains the definition of several related classes; so several classes are put together to compose several other related classes.

We can then conclude that parameterized packages have the expressive power of parameterized classes: a package containing just one class can be parameterized by package types containing a single interface each. This is achieved at the expense of extra syntax to define the packages and package types. That burden could be avoided by providing syntactic sugar for the special case of packages having only one class, and package types having only one interface. However, we do not bother to provide this facility here because, in practice, packages and package types very often contain several related types. This is definitely the case in Java, where even toy classes usually depend on the definition of specialized exception types [12]; and it is important to package together the class and the exception types, since usually one cannot be understood, reused or maintained without the other.

On the other hand, parameterized classes are not as expressive as parameterized packages. A parameterized class can impose (mutual) dependencies on its class parameters [16, 7, 17], but it cannot impose *subtype* dependencies between them[7], as shown in Section 2.1. Our experience with FOOPS [9] is that such conceptual and implementation dependencies between classes often arise when using the full power of parametrization for developing complex systems.

4.2 Type Systems

Parameterized classes are usually seen as user-defined type (class) constructors. This has a considerable impact on a language's type system. For instance, Pizza's type system [17] is much more complicated than Java's type system[8]. For extensions of Java [16, 17], the most unfortunate consequences of this impact are the subtle interaction between parametric and subtyping polymorphism[9], and the use of F-bounded polymorphism [5] to express either mutual dependencies between class

---

[6] Extending the Theta programming language with parameterized modules might have some drawbacks [7], but those definitely do not hold for PJava.

[7] Also, it seems it would be very awkward to do that.

[8] Though Pizza has several extra features, most complexity is due to parameterized classes.

[9] This leads to the question "are type constructors monotonic with respect to subtyping?", which should have a negative answer.

parameters [16, 7] or dependencies between a class parameter and its bound [17]. In Pizza, those consequences are aggravated by supporting implicit polymorphism with object creation and dynamic typing; that is usually modelled with existential types [6], which might not be familiar to Java programmers.

This kind of impact on Java's type system might have two undesirable practical consequences. First, Java programmers might have considerable difficulty for adapting to the extensions of Java. Second, type checking might be computationally expensive in those extensions.

Contrasting, PJava does not impact Java's type system. This is possible because language modifications are isolated to the level of packages. So there is no need for adding *type* constructors, but only for adding *package* constructors. However, it is necessary to provide a type system for the level of packages, where packages are values, "package types" are types, and "generic packages" are typed functions. So, instead of changing Java's type system, in PJava we introduce, at a different language level, an additional type system.

This approach actually minimizes the unfortunate practical consequences that would result from modifying Java's type system. This is possible because the package type system is very simple: it has no notion of subtyping and is based on the simple straightforward rules for matching arguments with parameters in the instantiation of generic packages. Furthermore, since package types can contain several mutually dependent classes, there is no need to express dependencies between parameters, or between parameters and their bound. Also, as parametrization and subtyping are at two different language levels, there is no problematic interaction between them.

4.3 Adaptive Software

For favouring software reuse, it is necessary to view software components differently in different contexts, since the contexts in which a component might be reused are usually quite different, and also different from the component's design and coding context. *Views* and *renaming* allow programmers to adapt software components to specific contexts without having to modify the components. Hence those features can actually adapt compiled components having no available source code; that is essential for adapting proprietary libraries for reuse.

Contrasting with PJava, other extensions of Java [16, 17] have no special mechanisms for adapting components. Instead, in those approaches a parameterized class can only accept as arguments classes having the same method *names* as those specified in its parameter types. In practice, this means that class writers should know the names of methods required by the generic classes that will be instantiated with the classes that they are writing!

This very unfortunate foreknowledge problem is totally contrary to the fundamental principle that software should be as adaptable as possible. It obviously hinders large scale, across corporations reuse. Worse, it also hinders small scale, inside corporation reuse. For example, consider a class `SortedList` parameterized by the type `T` of the elements that can be included in the list. This type must have a comparison method, say `boolean lt(T t)`, so that the list can be kept sorted. So in order to have a sorted list of bank accounts, accounts must have a method `boolean lt(Accountt a)`, which could well return true when the number of the account given as argument is lesser than the number of the account executing it. But what if in another context we need a list of accounts ordered in the opposite way, or simply a list ordered by owner's names? There is no way this could be done without views, unless we create *ad hoc* classes extending `Account` just for redefining `lt` and instantiating `SortedList`. There are several more unfortunate consequences of doing that[10] [10].

By using views, the situation described above would be naturally solved. In fact, in PJava we can not only establish that a package has many (package) types, but we can also establish that a package satisfies a type in more than one way[11]. That would be necessary for instantiating `SortedList` with the same argument, but in different ways.

Although views seem verbose, we think they are very important for adapting software components to new contexts and, consequently, enhancing reuse. Also, well thought rules for default matching can reduce much verbosity. Anyway, views are inherent to parameterization *in the large*, otherwise class names would have to match exactly in instantiations—a nonsense. Renaming is also inherent to our approach since two instantiations of the same generic package generate classes with the same name, and it is usually necessary to distinguish them.

5 Conclusions

In this paper we introduced PJava, a conservative extension of Java with support for parameterized *packages*. PJava's design was based on FOOPS [9, 2], minimizing language modifications and reducing impact on others aspects of Java. So, although we have significantly improved Java capabilities, we expect that Java programmers can easily adapt to PJava.

Comparing with related approaches for extending Java with generic *classes* [16, 17], PJava offers significant advantages: enhanced expressiveness and support for reuse and maintenance *in the large*; Java's type system is not impacted, since language modifications are isolated to the package level; foreknowledge problems are avoided due to much better support for adapting software to different contexts; parameter (package) types can be reused. Indeed, PJava's parametrization features can play an important role in restructuring Java libraries such that they can offer safer, more flexible and reusable components for programmers.

It would be desirable that package types could be structured and reused to define new, more complex, ones. As for (generic) packages, this should be possible through importation and parameterization. PJava's design could be easily adapted to support that, but this raised many implementation issues, since Java supports only a nontransitive import mode. We are working on a solution for future versions of the language.

Our translations of PJava into Java shows that PJava can be implemented in a relatively efficient way. Improved efficiency could be achieved by translating PJava directly into the JVM [13]. But, as noticed by the designers of Pizza [17], that would

[10] In Java, the situation is even worse since some classes—including `Integer`, `Boolean`, and `String`—are `final`: they cannot be extended.

[11] It might be useful to think of package types as specifications, and packages as implementations.

gain little in efficiency and lose much in clarity, since Java and the JVM are tightly coupled.

Much faster execution and reduced code size could be achieved by extending the JVM for dealing with generic code [16]. As the translation of parameterized packages into Java raises similar issues to the implementation of parameterized classes [16], it is likely that the extended JVM recently proposed [16] can also be used to efficiently implement parameterized packages. However, whereas this approach can still guarantee interoperability with Java code, it drastically reduces the number of available platforms, mainly web browsers and Java chips.

Acknowledgements

We would like to thank Prof. Joseph Goguen for his pervasive influence in this work. We are also grateful to members of the LinES research group, specially Augusto Sampaio, for listening and commenting on the work presented here, and to the anonymous referees who helped to improve this paper.

References

[1] Joseph Bank, Barbara Liskov, and Andrew Myers. Parameterized types and Java. Technical Report MIT LCS TM–553, Massachusetts Institue of Technology, May 1996.

[2] Paulo Borba and Joseph Goguen. An operational semantics for FOOPS. In Roel Wieringa and Remco Feenstra, editors, *International Workshop on Information Systems—Correctness and Reusability, IS-CORE'94*. Vrije Universiteit, Amsterdam, September 1994. A longer version appeared as Technical Monograph PRG-115, Oxford University, Computing Laboratory, Programming Research Group, November 1994.

[3] Renato Borges and Roberto Ierusalimschy. Módulos em linguagens orientadas a objetos. In *I Simpósio Brasileiro de Linguagens de Programação*, Belo Horizonte, Brazil, September 1996.

[4] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[5] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Fourth International Conference on Functional Programming and Computer Architecture*. ACM, September 1989.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys, ACM*, 17(4), December 1985.

[7] Mark Day, Robert Gruber, Barbara Liskov, and Andrew Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA '95 Conference Proceedings*, volume 30(10) of *ACM SIGPLAN Notices*, pages 156–168, 1995.

[8] Joseph Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, and José Meseguer. An introduction to OBJ3. In Jean-Pierre Jouannaud and Stephane Kaplan, editors, *Proceedings, Conference on Conditional Term Rewriting*, pages 258–263. Springer, 1988. Lecture Notes in Computer Science, Volume 308.

[9] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987.

[10] Joseph Goguen and Adolfo Socorro. Module composition and system design for the object paradigm. *Journal of Object-Oriented Programming*, 7(9), 1995.

[11] Joseph Goguen and David Wolfram. On types and FOOPS. In Robert Meersman, William Kent, and Samit Khosla, editors, *Object Oriented Databases: Analysis, Design and Construction*, pages 1–22. North Holland, 1991. Proceedings, IFIP TC2 Conference, Windermere, UK, 2–6 July 1990.

[12] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[14] David MacQueen. The implementation of Standard ML modules. In *ACM Conf. on Lisp and Functional Programming*, pages 212–23, New York, 1988. ACM Press.

[15] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[16] Andrew Myers, Joseph Bank, and Barbara Liskov. Parameterized types for Java. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, Paris, France, 15–17 January 1997.

[17] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.

[18] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.